East West University

PROJECT REPORT
FOR
Travel Planner Using Graphs

CSE246
Algorithms

Submitted By;

Jannatul Ferdaus Oishi	**2022-3-60-216**
Mahin Ahmed Meghla	**2022-3-60-152**

Submitted To;

Dr. Taskeed Jabid
Professor, Department of Computer Science and Engineering

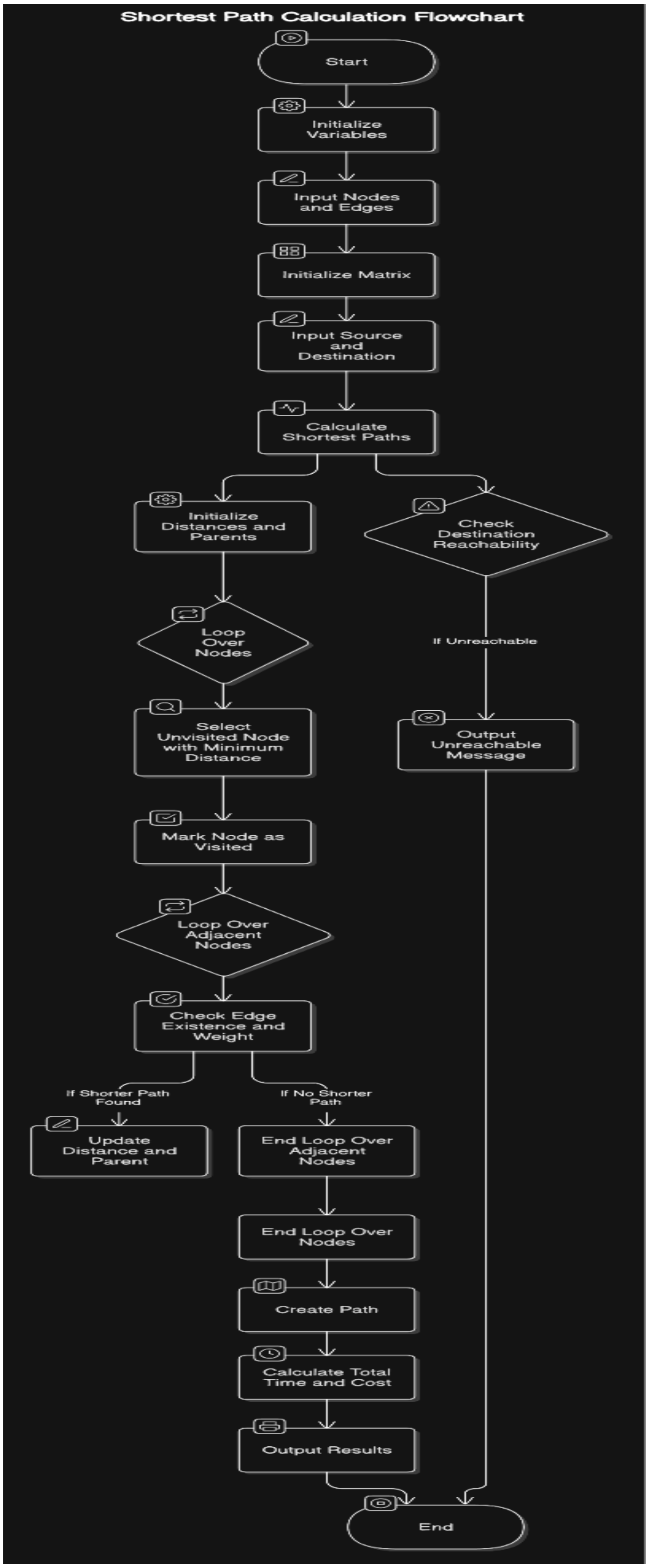
Date : 2nd February 2025

1. Introduction

This program implements a variant of Dijkstra's algorithm to find the shortest path between two nodes in a graph, where the graph's edges have three distinct attributes: distance, time, and cost. The user can choose which criterion to minimize when calculating the shortest path. This kind of program is useful for solving real-life problems where a user may need to optimize for multiple factors, such as finding the shortest route, minimizing travel time, or minimizing transportation costs.

The graph is represented as a weighted adjacency matrix, where each edge has values for distance, time, and cost. The program uses Dijkstra's algorithm to calculate the optimal path based on the user-selected criterion, then outputs the shortest path along with its total distance, time, and cost.

2. Flow Chart



Shortest Path Calculation Flowchart
Start
Initialize Variables
Input Nodes and Edges
Initialize Matrix
Input Source and Destination
Calculate Shortest Paths
Initialize Distances and Parents
Check Destination Reachability
Loop Over Nodes
If Unreachable
Select Unvisited Node with Minimum Distance
Output Unreachable Message
Mark Node as Visited
Loop Over Adjacent Nodes
Check Edge Existence and Weight
If Shorter Path Found
If No Shorter Path
Update Distance and Parent
End Loop Over Adjacent Nodes
End Loop Over Nodes
Create Path
Calculate Total Time and Cost
Output Results
End

3. Pseudo Code

```
function calculateShortestPaths(n, matrix, source, criteria):
    initialize dist and parent arrays
    set dist[source] = 0
    visited = array of false values

    for i from 0 to n-1:
        u = find the unvisited node with the smallest dist
        mark u as visited

        for each v adjacent to u:
            if edge exists based on criteria:
                weight = get weight based on criteria
                if dist[u] + weight < dist[v]:
                    dist[v] = dist[u] + weight
                    parent[v] = u

function createPath(destination):
    path = empty list
    while destination != -1:
        add destination to path
        destination = parent[destination]
    reverse path
    return path

main:
    initialize graph
    input source and destination
    calculateShortestPaths(n, matrix, source, "distance")
    if dist[destination] == INF:
        print unreachable message
    else:
        path = createPath(destination)
        calculate total time and cost
        display results
```

4. Types of Data

1. **Graph Representation**: An adjacency matrix is used to represent the graph, with each entry storing the edge attributes (distance, time, and cost) between nodes.
2. **Edge Structure**: A structure Edge is used to store the three values for each edge: distance, time, and cost.
3. **Arrays**: Arrays dist[] and parent[] are used to store the shortest distance from the source node to each node and the parent of each node in the path.

5. Time Complexity

The time complexity of Dijkstra's algorithm implemented in this code is O(n^2), where n is the number of nodes in the graph. This is due to the nested loops: one for selecting the unvisited node with the smallest distance and another for iterating through all adjacent nodes. The algorithm can be optimized to O((n + m) log n) using a priority queue, where m is the number of edges

6. Real-Life Applications

1. **Navigation Systems**: This program can be used in GPS-based navigation applications where users need to find the optimal route based on multiple factors like distance, time, and cost.
2. **Public Transport Systems**: For calculating the best public transport routes based on different factors (like time, cost, or distance).
3. **Ride-sharing Applications**: Ride-sharing services like Uber or Lyft can use similar algorithms to determine the best routes for drivers and passengers based on different criteria.
4. **Network Routing**: Helps in determining the most efficient path for data packets in computer networks.
5. **Game Development**: Used in pathfinding algorithms for characters in video games.

7. Future Scope

1. **Priority Queue Implementation:** Implementing Dijkstra's algorithm using a priority queue would improve the time complexity to O(m log n), making it more efficient for larger graphs.
2. **Handling Negative Weights:** The current implementation doesn't handle negative weights (for distance, time, or cost). Bellman-Ford algorithm could be used if negative weights are a possibility.
3. **More Sophisticated Cost Functions:** The cost function could be made more complex to include other factors like fuel consumption, number of turns, or road type.
4. **User Interface:** A graphical user interface could be developed to visualize the graph and the calculated paths.
5. **Dynamic Updates:** The program could be extended to handle dynamic changes in the graph for example ; road closures or traffic updates.

8. Conclusion

The provided code effectively demonstrates the implementation of Dijkstra's algorithm for finding the shortest path in a weighted graph. By allowing users to specify different criteria for optimization, the program showcases flexibility and adaptability. The analysis of time complexity and real-life applications highlights the algorithm's significance in various fields. Future enhancements can further improve its functionality and usability, making it a valuable tool for solving complex routing problems.